Interactive View-Dependent Rendering Of Large Isosurfaces

Benjamin Gregorski*† Mark Duchaineau* Peter Lindstrom* Valerio Pascucci*
Kenneth I. Joy†
*Center for Applied Scientific Computing, Lawrence Livermore National Laboratory
†Center for Image Processing and Integrated Computing, University of California, Davis

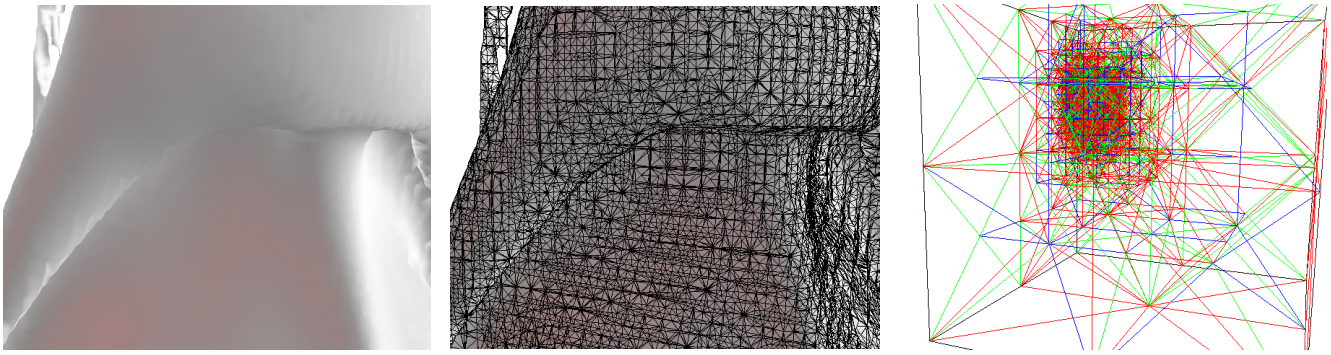

Figure 1: Closeup view of an isosurface feature in the mixing interface of two gases showing the texture mapped surface, underlying triangle mesh, and the adaptively refined tetrahedral mesh around the region of interest. Time step = 273, Isovalue = 206, Isosurface error = 1.5, 50K Triangles, rendered at 7 frames per second.

ABSTRACT

We present an algorithm for interactively extracting and rendering isosurfaces of large volume datasets in a view-dependent fashion. A recursive tetrahedral mesh refinement scheme, based on longest edge bisection, is used to hierarchically decompose the data into a multiresolution structure. This data structure allows fast extraction of arbitrary isosurfaces to within user specified view-dependent error bounds. A data layout scheme based on hierarchical space filling curves provides access to the data in a cache coherent manner that follows the data access pattern indicated by the mesh refinement.

CR Categories: I.3.5 [Computing Methodologies]: Computer Graphics—Computational Geometry and Object Modeling Curve, surface, solid, and object representations I.3.6 [Computing Methodologies]: Computer Graphics—Methodology and Techniques Graphics data structures and data types

Keywords: View-Dependent Rendering, Isosurfaces, Multiresolution Tetrahedral Meshes, Multiresolution Techniques

1 INTRODUCTION

The advent of high-performance computing has completely transformed the nature of most scientific and engineering disciplines making the study of complex problems from experimental and theoretical disciplines computationally feasible. Traditionally, with smaller and simpler data sets, researchers have developed *in-core* visualization and data exploration methods that work well on small or medium-scale datasets. They can quickly generate isosurfaces, and treat each isosurface independently. However today's impact problems of science and engineering require a different approach to address the increasingly difficult problems of organization, storage, transmission, visualization, exploration, and analysis associated with massive datasets.

We present a new algorithm for interactively extracting and rendering isosurfaces of large data sets in a view-dependent manner. Our algorithm generates isosurfaces "on-the-fly" using a view-dependent error measure, a recursive tetrahedral mesh refinement scheme, and a unique data layout scheme suitable for out-of-core visualization of large datasets.

Surface based level-of-detail techniques such as [5] and [21] extract a coarse isosurface and iteratively build a multiresolution surface model. For large volume datasets that contain topologically complex isosurfaces with millions and millions of triangles, these techniques need to be combined with out-of-core simplification techniques such as those developed by Lindstrom [11] and Lindstrom and Silva [13] in order to operate. In some cases, the storage requirements needed to extract, simplify, and visualize these surfaces can actually exceed those of the volume data from which they are derived [2]. Interactively visualizing these types of isosurfaces requires algorithms such as those developed by Duchaineau et al. [1, 2], that combine multiresolution representations, compression, and view-dependent optimizations. Surface based techniques are not suitable for visualizing volumes that contain a large number of isosurfaces that are important to the user because they must extract all of the interesting surfaces which would take far too much storage to be practical. On the other hand, volume based techniques, which extract and render the isosurfaces directly, do not require the precomputation of selected isosurfaces, and can easily switch between isovalues.

In this paper, we utilize the refinement of a tetrahedral mesh via longest-edge bisection to build a multiresolution hierarchy of a volume dataset. We combine coarse-to-fine and fine-to-coarse refinement schemes for this mesh to create an adaptively refinable tetrahedral mesh. This adaptive mesh supports a dual priority queue split/merge algorithm similar to the ROAM system [3] for view-dependent terrain visualization. It has fast coarsening and refine-

*{gregorski1,duchaine,pl,pascucci}@llnl.gov, †kijoy@ucdavis.edu

ment operations which allow for localized, incremental mesh updates, strict frame-to-frame triangle counts, progressive improvements of mesh quality, and guaranteed frame rates. The refinement scheme is coupled with a data storage scheme which aligns the data on disk and in main memory with the access pattern dictated by the mesh refinement. Sets of tetrahedra that share a common refinement edge are grouped into an aggregate structure called a diamond. Diamonds, as opposed to tetrahedra, function as the unit of operation in the mesh hierarchy and simplify the process of refining and coarsening the mesh.

At runtime, the split/merge refinement algorithm is used to create a lower resolution dataset that approximates the original dataset to within a given error tolerance. The error tolerance is a measure of how much an isosurface, extracted from the lower resolution dataset, deviates from the finest level isosurface. The error tolerance is measured in pixels on the view screen. The lower resolution dataset is a set of tetrahedra, possibly from different levels of the hierarchy, that approximates the volume dataset to within this isosurface error tolerance. This set of tetrahedra is free from cracks and T-intersections, and it defines a piecewise linear approximation of the original data. The isosurface is extracted from the tetrahedra in this lower resolution representation using linear interpolation.

In a preprocessing phase, we compute general information for each diamond that is used to drive the runtime mesh refinement. The following information is computed for the diamonds (Section 3.1):

1. The isosurface approximation error of the region enclosed by the diamond. (Section 5)

2. The min and max data values within the diamond including the diamond's boundary. The precomputed min/max ranges are used to quickly cull regions of the dataset that do not contain the isosurface.

3. The gradient vector at the center point of the diamond. The center point is also called the *split vertex* of the diamond. (Section 3.1) The precomputed gradient vectors are used to shade the isosurface using texture mapping.

The remainder of our paper is structured as follows: Section 2 reviews related work. Section 3 reviews longest edge bisection and introduces *parent* and *child* relationships for refining and coarsening the mesh. Section 4 describes the split/merge algorithm for refining and coarsening. Error metrics are described in Section 5. In Sections 6 and 7, we describe the data structures used to implement the split/merge refinement, and give an efficient, compact method to encode the mesh's structure. In Section 8, we discuss our data layout scheme. Our results are shown in Section 9.

2 PREVIOUS WORK

The refinement of a tetrahedral mesh via longest edge bisection is described in detail in several papers. In Zhou et al. [23], a fine-to-coarse merging of groups of tetrahedra is used to construct a multilevel representation of a dataset. Their representation approximates the original dataset to within a specified tolerance and preserves the topology of the finest level mesh. For larger datasets, this fine-to-coarse strategy is not practical because storing the finest level mesh would require too much memory.

An improved algorithm for preserving the topology of an extracted isosurface is presented by Gerstner and Pajarola [8]. This algorithm is combined with a coarse-to-fine splitting of tetrahedra to extract topology preserving isosurfaces or to perform controlled topology simplification. Rendering of multiple transparent isosurfaces and parallel extraction of isosurfaces are presented by Gerstner [6] and by Gerstner and Rumpf [7]. Both of these algorithms extract the isosurfaces from the mesh in a coarse-to-fine manner. In Roxborough and Nielson [20], the coarse-to-fine refinement algorithm is used to model 3-dimensional ultrasound data. The adaptivity of the mesh refinement is used to create a model of the volume that conforms to the complexity of the underlying data.

View-dependent extraction of isosurfaces utilizes multiresolution representations to extract surfaces that satisfy certain visual requirements. These requirements are usually based on the distance of the surface from the viewpoint, the position of the surface relative to the view-frustum, and the occlusion of the surface. Duchaineau et al. [3] control refinement using the screen space projection error, view-frustum culling, and line of site corrections. Occlusion culling supplements view-frustum culling by finding areas within the visible region that cannot be seen. In Livnat and Hansen [15], a hierarchical visibility test is used to determine regions of the volume that are occluded. The volume is decomposed using an octree, and the visibility test is performed using hierarchical tiles based on coverage masks (see Greene [9]). A shear warp transformation is used to perform the screen space projection. Their visibility algorithm requires that the octree be traversed from front to back. Zhang et al. [22] divide a large dataset into a set of independent blocks. They use ray casting from the viewpoint into the volume to determine a subset of these blocks that are occluders. These initial occluding blocks are rendered to create an occlusion mask that shows which screen pixels are covered. The remaining blocks are traversed and rendered if they are not completely occluded. Unlike [15], this last rendering step does not traverse the blocks in a front-to-back order. This ray tracing approach is also used in [14] to find an initial set of voxels from which to propagate the isosurface. The algorithm starts extracting the isosurface from these seed sets, and detects when the surface folds back on itself and becomes occluded. In our algorithm, we use the screen space projection error of the isosurface and view-frustum culling to control the view-dependent refinement.

In this work, we are focused on developing algorithms for level-of-detail based, interactive exploration of large, complex isosurfaces. Surface based methods such as [2, 21] construct level-of-detail surface models that are suitable for interactive view-dependent rendering. Volume based techniques such as [15, 22] speed up the search for cells that contain the isosurface and cells that do not need to be rendered, but they extract the isosurface from the finest level cells. Our algorithm differs from these approaches by utilizing a level-of-detail volumetric model which extracts the isosurface from coarser representations of the volume that meet certain requirements. Isosurface extraction techniques based upon level-of-detail allow the isosurface to be progressively refined over time, see for example [4, 18]. Visualizing large, complex isosurfaces often requires the ability to fly through the dataset and closely inspect areas of interest. Level-of-detail methods that support strict triangle counts per frame for efficient rendering, progressive improvement of mesh quality to provide guaranteed frame rates, and coherent access to data to minimize memory faults are well suited to this task.

3 LONGEST EDGE BISECTION

In this section we review longest edge bisection and establish terminology. In this scheme, a tetrahedron is described by a *level* and a *phase*, with 3 phases at each level. The bisection begins at level 0, phase 0 with an initial configuration of a cube divided into 6 tetrahedra around a major diagonal. Figure 2 illustrates the three phases of the refinement process. After three refinements, the level is incremented by 1. After *n* refinements, the phase is $n \bmod 3$ and the level is $\lfloor n/3 \rfloor$. The *split edge* of a tetrahedron is its longest edge. In each phase, a tetrahedron is subdivided into two *children* at the midpoint of the split edge. This midpoint is called the *split vertex*.

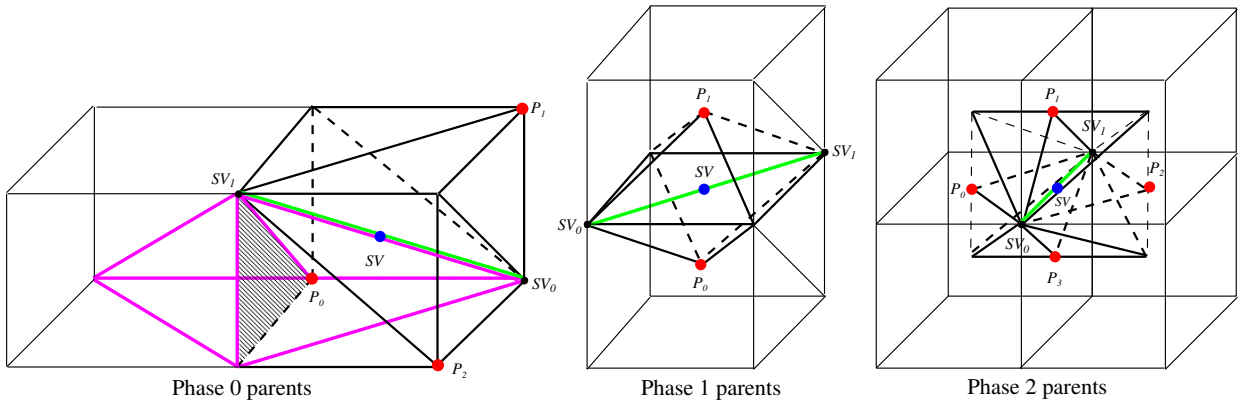

Figure 3: Parent diamonds: Phase 0 parents are phase 2 diamonds from the level $L-1$, phase 1 parents are located at cube centers, and phase 2 parents are located at face centers. The split edge is (SV_0, SV_1) (shown in green), the split vertex is SV (blue), and the parents are shown as P_0, P_1, P_2, and P_3 (red). The magenta tetrahedron is a tetrahedron in diamond P_0. The shaded triangle shows how it is split into two phase 0 tetrahedra.

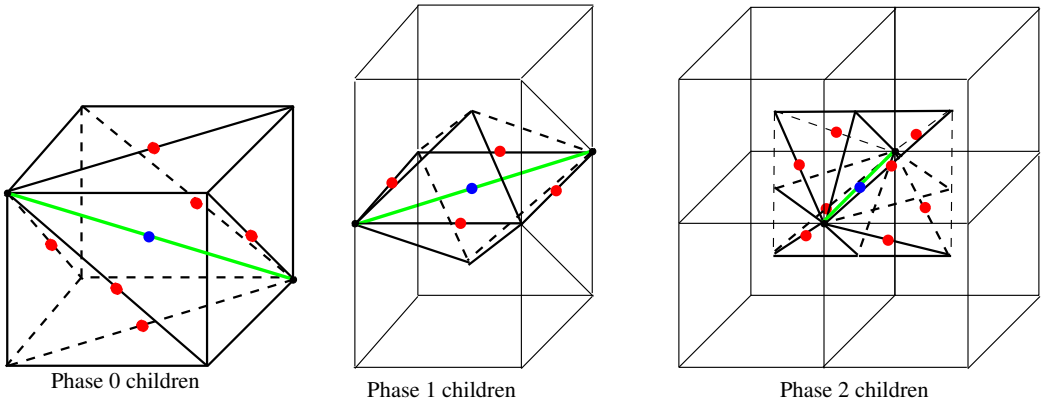

Figure 4: Child diamonds: Phase 0 children are located on the faces of a cube, phase 1 children are located on the centers of the edges of the face containing the split edge, and phase 2 children are the phase 0 diamonds from level $L+1$ that touch the diamond's split edge.

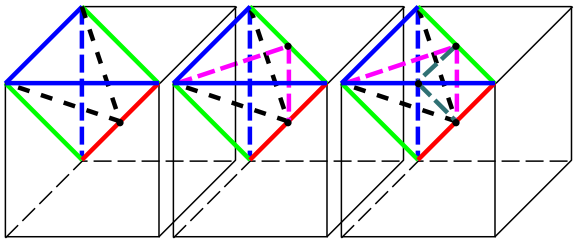

Figure 2: Three phases of refinement for a single tetrahedron of the initial configuration.

3.1 Diamonds

Tetrahedra are grouped into diamonds to simplify the refinement process and to ensure continuity of isosurfaces generated from the mesh. When a tetrahedron is split, all the tetrahedra that share its split edge must also be split. A group of tetrahedra that share a split edge is called a *diamond*. The split edge and split vertex of a diamond are defined as the common split edge and split vertex of its tetrahedra. All diamonds in the mesh can be uniquely identified by their split edge or split vertex. Phase 0, phase 1, and phase 2 diamonds are shown in Figures 3 and 4. Each point in the dataset, except for the corner points of the original cube, corresponds to the split vertex of one diamond because each point is introduced by the splitting of a diamond. By grouping tetrahedra into diamonds, we can easily locate all of the tetrahedra around a split edge. Splitting a diamond is equivalent to splitting all of the tetrahedra in the diamond. All tetrahedra within a diamond have the same level and phase. Table 1 lists the number of tetrahedra, their phase, and level for each diamond.

Phase	Tets(Phase,Level)	Parents(P,L)	Children(P,L)
0	6(0,L)	3(2,L-1)	6(1,L)
1	4(1,L)	2(0,L)	4(2,L)
2	8(2,L)	4(1,L)	8(0,L+1)

Table 1: Number, phase, and level of tetrahedra, parents, and children for the three different diamonds. L is the *level* of the diamond.

The *type* of a diamond is determined by its split edge (SV_0, SV_1), where SV_0 and SV_1 are the vertices on the split edge. Starting from the initial configuration of six tetrahedra in a cube, there are 26 different direction vectors (i.e. diamond types) for the split edge; there are 8 directions for the phase 0 diamonds, 12 for the phase 1 diamonds (4 each on the XY, XZ, and YZ planes), and 6 for the phase 2 diamonds. The type of a diamond is used to efficiently encode the structure of the mesh (Section 6) including the

location of parent and child diamonds (Section 3.2).

3.2 Parent And Child Diamonds

Given a diamond D, the parents of D are the diamonds that must be split to create D's tetrahedra. Figure 3 shows the parents for each diamond. The diamonds that are created when D is split are called D's *children*. Figure 4 shows the children for each diamond. In these figures, a diamond is indicated by its split vertex. The parent and child information is summarized in Table 1.

4 SPLIT/MERGE REFINEMENT

The tetrahedral mesh supports the dual queue split/merge refinement strategy similar to that described by Duchaineau et al. [3]. This strategy provides more frame-to-frame coherence than a coarse-to-fine only algorithm. It allows us to control the triangle count per frame, and to effectively cache previously computed geometry to minimize expensive interpolation calculations. In most interactive applications, the viewing position does not change significantly between consecutive frames. In frame $i+1$, many diamonds from frame i will have a view-dependent error that is still within the error tolerance. These diamonds can be reused in frame $i+1$. A small fraction of the diamonds must be split or merged to satisfy the error tolerance. By starting the refinement process for frame $i+1$ with the mesh from frame i instead of the base mesh, fewer splits and merges are performed.

The *current mesh* is a set of tetrahedra that approximates the volume dataset to within a certain view-dependent error bound. The mesh is generated using two priority queues. The split queue holds the diamonds containing the tetrahedra of the current mesh. The merge queue holds the diamonds that have been split and whose children have not been split (i.e. diamonds with children but no grandchildren).

At frame 0, the split queue is initialized with the base configuration of six tetrahedra (the root diamond), and the merge queue is empty. At each frame, given a view-dependent error tolerance E, the following steps are taken:

1. Diamonds not within the view frustum are marked as *invisible* and diamonds that do not contain the isosurface are marked as *empty*; they are assigned a view-dependent error of zero. View-dependent errors are recomputed for all other diamonds in the split and merge queues.

2. Diamonds in the split queue whose error is greater than E are split. Diamonds in the merge queue whose error is less than E are merged. *Invisible* and *empty* diamonds in the split queue are never split. In the merge queue, they are the first diamonds to be merged.

3. The refinement process is stopped when all diamonds in the split queue have an error below E and all diamonds in the merge queue have an error above E, or when the time allowed for processing the current frame has elapsed.

4. The isosurface is extracted from the tetrahedra that belong to the visible, non-empty diamonds in the split queue.

A diamond D is split by splitting all of its tetrahedra, and inserting the child tetrahedra into the split queue. A tetrahedron is placed into the split queue by creating an entry for its diamond and adding the tetrahedron to the diamond. There is only one entry for a diamond in the split queue. When some of the tetrahedra in a diamond do not exist (i.e. they are not in the current mesh), it is necessary to create them before the diamond can be split. This situation is shown in 2D in Figure 5. The tetrahedra are created by splitting the parents of D that have not been split. When all the parents and tetrahedra of D have been split, D is removed from the split queue and added to the merge queue.

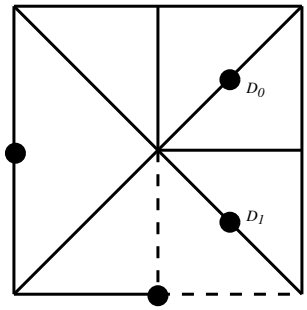

Figure 5: Diamond D_0 has two triangles in the mesh. Diamond D_1 has two triangles, one of which in the mesh. The triangle not in the mesh is shown with the dashed lines. This is a 2D analogy of the 3D tetrahedral mesh.

Merging a diamond is done by merging all of its tetrahedra, and adding them to the split queue. A tetrahedron is merged by removing its two children from the split queue. A tetrahedron is removed from the split queue by locating its diamond's entry in the split queue and removing it from the diamond. When a tetrahedron is removed from the mesh, its diamond is checked to see if all the tetrahedra of the diamond have been removed from the queue. If so, the diamond is removed from the split queue. Lastly, the diamond's parents are checked to see if they can be added to the merge queue. A diamond can be added to the merge queue only if all of its children are in the split queue.

4.1 Modifying The Isovalue

When the isovalue is changed by the user, the new isosurface can be extracted by starting at the root diamond or starting from the current mesh. In the first case, the split and merge queues, hash tables, and isosurface are invalidated and initialized with the root diamond. The split/merge refinement is then started from this initial configuration. In the second case, the split queue, merge queue, and hash tables remain the same, and the old isosurface is thrown away. The diamonds in the split and merge queues are checked to determine if they contain the new isovalue. Diamonds that do not contain the isovalue are marked as empty and given an approximation error of zero. Isosurface errors are computed for those diamonds that contain the new isovalue. The split/merge refinement continues from this new configuration. Diamonds that contain the new isosurface will be refined if their error is too large and coarsened if their error is too small. Diamonds that contained the old isosurface, but do not contain the new isosurface, will be merged because they are no longer needed to represent the volume. The effectiveness of both of these methods depends on the locality of the old and new isosurfaces in the mesh hierarchy. Starting from the current configuration makes sense if they are close together, and starting from the top makes sense if they are far apart.

5 ERROR METRICS

Each diamond in the mesh has an associated approximation error, isosurface error, and view-dependent error. The approximation error e_a for a tetrahedron T is the maximum difference between the linear approximation over T that interpolates the values at $T's$ vertices and the actual data values for the points inside T and on its

boundary (i.e. faces, edges, and vertices). The approximation error for a diamond D is the maximum of the approximation errors of its tetrahedra. Leaf tetrahedra and leaf diamonds have an approximation error of zero.

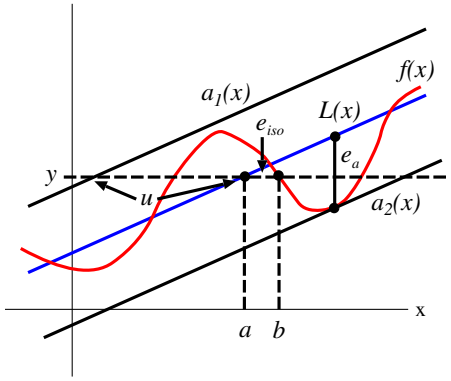

Figure 6: Isosurface error calculation in 1D.

The isosurface error of a tetrahedron T is the maximum deviation of an isosurface generated using the scalar values at the vertices of T from the true isosurface passing through T. This calculation is illustrated in Figure 6 for the one-dimensional case. The original function is $f(x)$ and it is approximated by $L(x)$. The upper and lower bounds on the approximation, given by the approximation error e_a, are $a_1(x)$ and $a_2(x)$. For a given function value y, the isocontour using $L(x)$ occurs at point a where $y = L(a)$, while the true isocontour using $f(x)$ occurs at the point b where $y = f(b)$. The error in the isocontour is given by:

$$e_{iso} = |a - b| \tag{1}$$

An upper bound u for the isosurface error can be computed by:

$$u = e_a/k \geq e_{iso}, \tag{2}$$

where k is slope of the linear approximation L. As f approaches a vertical line the slope of L increases, and f is approximated with increasing accuracy by L. As the slope of f decreases, the isocontour approximation a and the true isocontour b can be far apart even if e_a is small. In higher dimensions, the slope of the approximation translates to the magnitude of the gradient. In three-dimensions, this is the gradient of the field through a tetrahedron as given by the linear function that interpolates the values at the tetrahedron's vertices. The isosurface error is clamped at the physical size of the tetrahedron because the isosurface drawn through a tetrahedron can never be outside the tetrahedron's boundaries. The isosurface error for a tetrahedron T is given by:

$$e_{iso}(T) = \min(e_a/\|\nabla f(T)\|, diam(T)), \tag{3}$$

The isosurface error $e_{iso}(D)$ for a diamond is:

$$e_{iso}(D) = \max(e_{iso}(T), \forall T \in D). \tag{4}$$

The view-dependent error of a diamond is the projection of its isosurface error onto the view screen. This projection is done by creating a sphere at the diamond's split vertex of radius $e_{iso}(D)$ and projecting this sphere onto the view screen. The size of the projected sphere (i.e. width or height in pixels) is the view-dependent error. Details on view-dependent error metrics can be found in Hoppe [10], Lindstrom and Pascucci [12], and Luebke and Erikson [16]. All of these error metrics are easily incorporated into our refinement strategy.

6 MESH ENCODING

The mesh structure can be encoded in a very compact manner assuming that the data points lie on a $(2^n + 1) \times (2^n + 1) \times (2^n + 1)$ grid. In this case, the offsets, relative to the split vertex of the diamond, to compute the tetrahedron vertices, parents, and children of a diamond are all powers of two. Data that do not lie on such a grid can either be resampled to lie on a grid of the proper size or embedded in a *virtual grid* of the proper size.

Since each data point corresponds to a diamond, we represent a diamond using an (i, j, k) index. This index corresponds to the index used to access the precomputed diamond information and data values if they were stored in a C-style 3-dimensional array. The vertices defining the split edge of a diamond are encoded in a single byte as an offset vector from the split vertex. For example, the split edge with $SV_0 = (64, 64, 0)$ and $SV_1 = (64, 0, 64)$ has the vector $(0, -64, 64)$ and split vertex $(64, 32, 32)$. Dividing this vector by 64 yields $(0, -1, 1)$. These values are stored as 2 bit quantities in a single byte. SV_0 and SV_1 are computed by rescaling the vector and adding/subtracting it from the split vertex. In this example, $(0, -1, 1)$ is rescaled to $(0, -32, 32)$. The rescaling factor is easily determined from the level of the diamond. For a mesh with l levels, the scaling factor for a diamond at level j is given by 2^{l-j-1}. The split edge encodings are stored in a lookup table and accessed at runtime based upon the type of the diamond. Since a diamond is identified by its split vertex, the vertices on the split edge can be computed by knowing the diamond's type and level. The parents, tetrahedra, and children of a diamond are encoded and stored in the same manner as the split edge. For any diamond, the (i, j, k) index for a parent, child or vertex of a tetrahedron can be computed from the diamond's split vertex and the proper encoding. There is one set of encodings for each of the 26 types of diamonds.

Encoding the mesh in this manner allows us to quickly compute the (i, j, k) index of a diamond's parents and children. Instead of storing pointers to the parents and children of a diamond, we store all of the diamonds in a hash table and use the (i, j, k) index of the split vertex to locate a diamond. In the case of a phase 2 diamond, this saves twelve pointers (4 parents, 8 children) or 48 bytes per diamond. Since each (i, j, k) index corresponds to a data point, we can quickly compute indices for the vertices of a tetrahedron and use them to get the data values needed to extract the isosurface.

7 DATA STRUCTURES

We precompute the isosurface approximation error, min and max scalar field values, and gradient vector at the split vertex for each diamond in the hierarchy. Gradients can be computed at runtime and stored in a hash table; however, our data layout algorithm (Section 8) makes computing gradients via central differences expensive and so the gradients are precomputed. Assuming byte scalar field data, floating point errors and floating point gradient components, 18 additional bytes are required per input value. A 1024^3 (1 Gb) dataset would be inflated to an enormous 19 Gb dataset.

In order to reduce the size of the precomputed data, the isosurface errors are compressed on a logarithmic scale on a per-level basis and represented in six bits. The gradient vectors are quantized on a unit cube using fourteen bits. In addition, we use an iterative relaxation process to smooth the gradient vectors which are computed directly from the byte datasets we use. The min and max values for a diamond D are compressed in relation to a diamond S that completely contains D. A diamond S contains a diamond D if the polyhedron for D is completely enclosed by the polyhedron for S. The tetrahedra created by recursively refining D's tetrahedra are all contained within S's polyhedron. Either of the two diamonds whose split vertices are the vertices of $D's$ split edge can be used for S. This is illustrated in Figure 7.

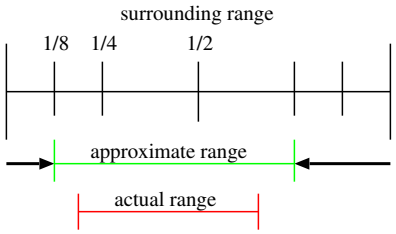

Figure 7: The min/max values of a diamond are encoded relative to the min and max values of an enclosing diamond using 4 bits (2 each for min/max) to encode 0/8, 1/8, 1/4, or 1/2 of the enclosing interval. (The offset 3/8 is not encoded.) In this example min = 1/8, max = 1/4.

Using these data structures, the precomputed information for each diamond is stored in three bytes. A 1 Gb dataset is inflated to 4 Gb instead of 19 Gb. Error values and gradients are found at runtime using lookup tables. Since the errors are encoded in 6 bits, the table for the error values contains $2^6 \times n$ floating point values where n is the number of levels in the mesh. For a 512^3 dataset, n equals 9. The gradient table contains 3×2^{14} floating point values.

The split and merge queues are implemented as hash tables using a fixed number of buckets and chaining to handle collisions. Each bucket corresponds to a range of the projected screen space error as measured in pixels. Each entry in the bucket corresponds to a diamond whose screen space error falls within the bucket's range. The buckets are not sorted internally by error value. Hash tables can be used instead of priority queues because it is not necessary to split the diamond in the split queue with the highest error, or to merge the diamond in the merge queue with the lowest error. Instead it is sufficient to split a diamond whose error is greater than the current tolerance and to merge a diamond whose error is less than the current tolerance. Hash tables with $O(1)$ operations provide better performance than a priority queue with $O(\log n)$ operations. A separate hash table, the queue hash table, is used to map diamond indices to their entries in the queue. There is one hash table for the split queue and one hash table for the merge queue. This second hash table is necessary because the split and merge queues are ordered by view-dependent error. In order to quickly locate a specific diamond in either queue, we need to be able to access the queue based upon the (i, j, k) index of the diamond. Accessing the diamonds in the queues based on view-dependent error would require computing the view-dependent error, locating the bucket that the diamond is in, and then traversing the bucket to get the appropriate entry.

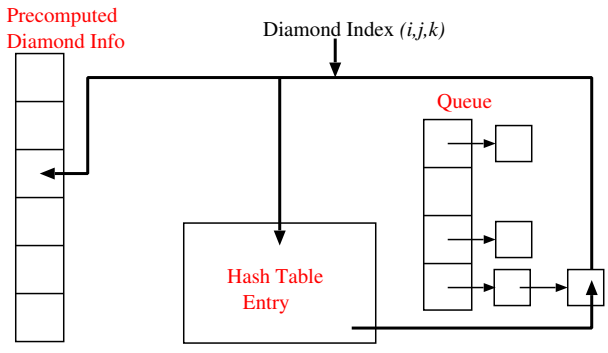

Figure 8: Relationship between precomputed data, queue entries, and queue hash table.

The data structures are illustrated in Figure 8. The hash table maps a diamond index to an entry in the queue. The diamond index associated with the queue entry maps back to the precomputed diamond information and the same hash table entry. When a tetrahedron is added or removed from the mesh, its diamond's index is used to locate the corresponding entry in the split queue via the split queue's hash table. Each diamond in the split queue contains flags indicating which of its tetrahedra are actually in the current mesh. The reason for these flags is shown in Figure 5. Each bucket entry in the split and merge queues stores the diamond's level, (i, j, k) index, isosurface and view-dependent errors, and *invisible* and *empty* bits.

When a tetrahedron is added to the split queue, the isosurface passing through it is computed and stored in the *geometry cache*. The geometry is cached in an array so that it is in a contiguous region of memory. New geometry is appended to the end of the array. Geometry is removed from the cache by replacing the removed geometry with geometry at the end of the array. A hash table is used to map a diamond to the geometry cache entries associated with its tetrahedra. This caching method duplicates normals and vertices along edges. Its advantage is that it has better memory coherence than hash table based caches which store the vertices and normals on a per-edge basis (see Gerstner and Rumpf [7]). On newer graphics hardware, storing the geometry in a contiguous region of memory allows one to stream the data from memory to the graphics card for improved rendering performance. The mesh is drawn by traversing the geometry cache.

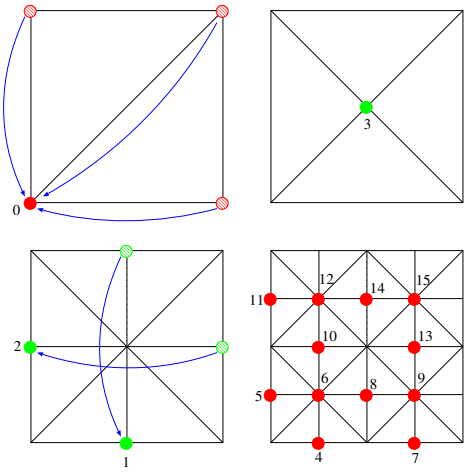

Figure 9: New data points required at each refinement level in 2D. The arrows indicate the wrapping of data values assuming periodic boundary conditions.

8 MEMORY LAYOUT

When visualizing very large datasets, memory performance is a key bottleneck that must be overcome to achieve interactivity. In order to improve cache performance and effectively utilize the available memory bandwidth, we arrange our data on disk and in memory in a hierarchical z-order layout which follows the data ordering indicated by the mesh refinement.

Figure 9 shows how the mesh refinement algorithm accesses the data. Starting with the root configuration in the upper left, the dots indicate which data points are introduced at each refinement step. The numbers indicate the order in which the points are stored in a one-dimensional array. The dataset is stored first by level-of-detail (i.e. quadtree or octree level) and then by geometric proximity within each level. This layout scheme assumes that the dataset ex-

	Processors	Memory Usage (Mb)	
	R10K (250 Mhz)	Physical	Total
Preprocess	8	800	800
Runtime	1	150	690

Table 2: Memory and processor usage for the preprocessing and runtime phases for a 512^3 dataset. Runtime measurements are taken for an isosurface with 70K triangles and an error of 1.5.

Time(s)	Operation	# Elements	Elem/Sec
0.07	Cull/Priority	49K - 77K	700K - 1.1M
0.06	Drawing	63K	1.05M
0.01	Split/Merge	10-40	1000 - 4000

Table 3: Timings results for algorithm sections.

hibits periodic boundary conditions which is a valid assumption for the datasets that we are working with (i.e. for a 129^3 dataset indices 0 and 128 map to the same location). This data layout scheme and its performance benefits are detailed in [12] and [19]. Storing the data in this manner improves the coherence of the data access which is essential when working with large datasets. The original dataset and the information computed in the preprocessing phase of our algorithm are stored on disk in this manner. The data is mapped from disk to main memory at runtime using the Unix **mmap** command. The mmap command establishes a mapping between a process's address space and a virtual memory object represented as a disk file. It provides us with a basic out-of-core paging algorithm. This allows us to keep in memory the data that is currently being used by the split/merge process and the isosurface extraction process.

9 RESULTS

We have tested our methods on an SGI Onyx with 44 250 MHZ R10K processors and IR2 graphics boards. At runtime the algorithm uses one processor and one graphics pipe. The preprocessing was done in parallel on the same machine. Memory and processor usage for the preprocessing and runtime phases is shown in Table 2. Resident memory refers to the actual physical memory used. It includes memory used by the data structures and by the regions of the dataset that have been paged in from disk. Total memory refers to the address space currently assigned to the program. Preprocessing a 512^3 dataset takes 3.1 hours, and the final dataset size is 537 Mb.

Our test dataset is the Gordon Bell Prize winning simulation of a Richtmyer-Meshkov instability in a shock tube experiment [17]. The full resolution dataset consists of 274 time steps with each time step divided into a grid of $8\times8\times15$ bricks where each brick consists of $256\times256\times128$ byte data values for a total time step resolution of $2048\times2048\times1920$ byte data values. A full resolution isosurface of the mixing interface produces a mesh with 460 million triangles. In our examples, we are looking at isosurfaces of entropy values calculated as two fluids mix over time. Our examples are from 512^3 chunks cropped from the full resolution dataset. Figure 10 shows how the isosurface refines around the viewpoint and coarsens away from the viewpoint. Figure 1 shows a closeup view of a feature in the mixing interface at time step 273. The ability to zoom in on regions of the dataset and refine the isosurface allows one to closely inspect the features of the mixing process. These images show how the mesh adaptively refines around the viewpoint. Figure 11 shows an isosurface similar to the one shown in Figure 1 at different screen space errors. The isosurface representing the mixing interface contains a large number of topological components and small features. In the two lower resolution surfaces, the small feature in the top left is not preserved.

Table 3 shows the performance measurements for the visibility culling and priority recomputation, split/merge refinement, and rendering sections of our algorithm. The time for culling and priority recomputation depends on the number of computations and the memory performance of the hash table. We can perform about 700K - 1.1M computations a second. Rendering at 7 FPS (0.14s per frame) and allowing at most half of the frame time for culling and priority recomputation, we are allowed 49K - 77K computations per frame. Limiting the triangle count in the extracted isosurface to around 50K triangles gives us roughly 65K - 85K diamonds in the queues. We recompute the visibility information for all diamonds in the split and merge queues, and we recompute the priorities for all visible, non-empty diamonds in the queues. This is an expensive operation and can be improved using hierarchical, deferred priority recomputation. The split/merge performance is determined by the number of recursive splits and the coherency of the data access. Merges only have to look at children and parents and perform $O(1)$ lookups to find them. Splits look at children and parents and may have to look at $O(n)$ diamonds where n is the number of levels in the tree. To test the split/merge performance, we fixed the time for doing splits and merges to 0.01s. The algorithm performs around 1000 - 4000 updates per second or 10 - 40 updates per frame. The time for drawing depends on the number of elements drawn. In immediate mode, the SGI graphics system can draw 50K triangles at a rate of 1.05M triangles per second which is about 20 frames per second. At 7 FPS with 0.06s for drawing, this gives us at most 63K triangles per frame. These restrictions on the triangle count, queue sizes, and mesh updates per frame, allow us to render 5 - 7 frames per second or about 250K - 350K triangles per second.

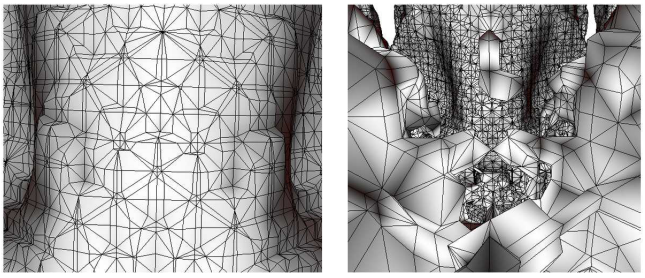

Figure 10: Closeup view of a mixing feature. Time Step = 273, Isovalue = 186, Isosurface error = 0.5. On the right, a zoomed out view shows the portion of the isosurface culled by the view frustum.

10 CONCLUSIONS

We have presented an algorithm for quickly extracting and rendering isosurfaces from large volume datasets. Our algorithm uses a multi-resolution tetrahedral mesh based on edge bisection, extending it to support adaptive refinement and coarsening. We have shown an efficient way to encode the tetrahedra, parents, and children of the mesh structure so that the mesh can be represented compactly and computed using efficient integer operations. The implementation of the dual queue split/merge algorithm utilizes this new encoding of the mesh through the addition of a queue hash table which enables the queues to be accessed by view-dependent error and diamond indices. Our algorithm is very extensible and easily integrates with other optimizations such as deferred priority recomputation, front-to-back traversal for transparent rendering and occlusion culling, topology preservation and simplification, and parallelization.

The size and complexity of datasets such as the Richtmyer-Meshkov simulation present great visualization challenges. Our future work is focused on extending our algorithms to be able to handle the full size datasets from these simulations in interactive

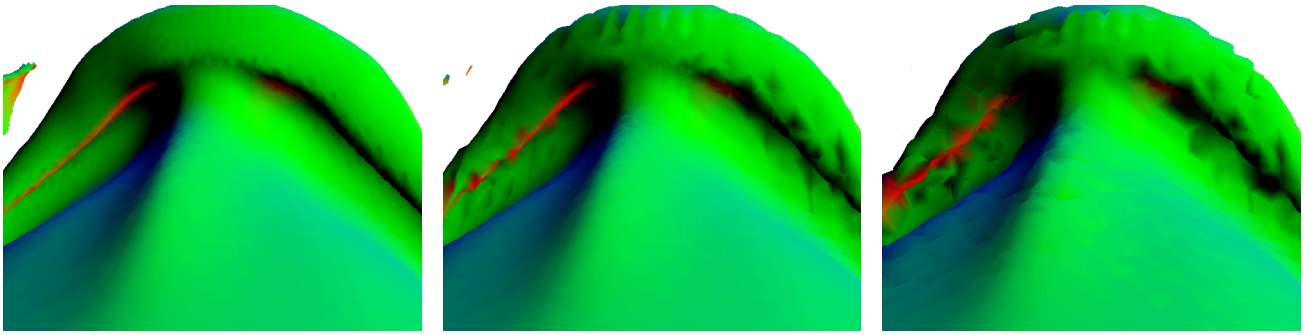

Figure 11: Isosurface with varying screen space error at Time Step = 273, Isovalue = 213. From left to right: Error = 0.56, 95K Triangles; Error = 1.7, 30K Triangles; Error = 2.7, 13K Triangles.

applications. Runtime processing of these massive datasets requires parallelization of both the refinement process and the rendering process. This can either be done on large shared memory machines or large clusters of commodity workstations. Both techniques require efficient data paging schemes to move data between processors and machines. The visualization of time varying data presents an even bigger challenge for these large datasets. Extending our algorithms to time varying data requires time varying encoding and compression of the data as well as fast decoding and decompression to update the mesh as quickly as possible.

11 ACKNOWLEDGEMENTS

This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48. We also thank the people at the Center for Image Processing and Integrated Computing (CIPIC) at the University of California Davis for all of their help and suggestions with this work.

REFERENCES

[1] Mark A. Duchaineau, Martin Bertram, Serban Porumbescu, Bernd Hamann, and Kenneth I. Joy. Interactive Display Of Surfaces Using Subdivision Surfaces And Wavelets. In T.L. Kunii, editor, *Proceedings of 16th Spring Conference on Computer Graphics, Comenius University, Bratislava, Slovak Republic*, pages 22–34, 2001.

[2] Mark A. Duchaineau, Serban Porumbescu, Martin Bertram, Bernd Hamann, and Kenneth I. Joy. Dataflow And Re-Mapping For Wavelet Compression And View-Dependent Optimization Of Billion-Triangle Isosurfaces. In G. Farin, H. Hagen, and B. Hamann, editors, *Hierarchical Approximation and Geometrical Methods for Scientific Visualization*. Springer-Verlag, Berlin, Germany, (to appear)., 2002.

[3] Mark A. Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. ROAMing Terrain: Real-time Optimally Adapting Meshes. In *IEEE Visualization '97*, pages 81–88. IEEE Computer Society Press, October 1997.

[4] Klaus Engel, Rdiger Westermann, and Thomas Ertl. Isosurface Extraction Techniques For Web-Based Volume Visualization. In *Proceedings of IEEE Visualization 1999*, pages 139–146. IEEE Computer Society Press, 1999.

[5] Marcel Gavriliu, Joel Carrance, David E. Breen, and Alan H. Barr. Fast Extraction Of Adaptive Multiresolution Meshes With Guaranteed Properties From Volumetric Data. In T. Ertl, K. I. Joy, and A. Varshney, editors, *Proceedings Visualization 2001*, pages 295–302. IEEE Computer Society Press, 2001.

[6] T. Gerstner. Fast Multiresolution Extraction Of Multiple Transparent Isosurfaces. In Ronald Peikert David S. Ebert, Jean M. Favre, editor, *In Data Visualization 2001 Proceedings of VisSim 2001*, Annual Conference Series. Springer-Verlag.

[7] T. Gerstner and M. Rumpf. Multiresolution Parallel Isosurface Extraction Based On Tetrahedral Bisection. In M. Chen, A. Kaufman, and R. Yagel, editors, *Volume Graphics*, pages 267–278. Springer-Verlag, 2000.

[8] Thomas Gerstner and Renato Pajarola. Topology Preserving And Controlled Topology Simplifying Multiresolution Isosurface Extraction. In T. Ertl, B. Hamann, and A. Varshney, editors, *Proceedings of IEEE Visualization 2000*, pages 259–266. IEEE Computer Society Press, 2000.

[9] Ned Greene. Hierarchical Polygon Tiling With Coverage Masks. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 65–74. Addison Wesley. New Orleans, Louisiana, 04-09 August 1996.

[10] Hugues Hoppe. View-Dependent Refinement Of Progressive Meshes. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 189–198. Addison Wesley, 1997. Los Angeles, California, 03-08 August 1997.

[11] Peter Lindstrom. Out-Of-Core Simplification Of Large Polygonal Models. In Kurt Akeley, editor, *SIGGRAPH 2000 Conference Proceedings*, pages 259–262. ACM Press / Addison Wesley, 2000.

[12] Peter Lindstrom and Valerio Pascucci. Visualization Of Large Terrains Made Easy. In T. Ertl, K. Joy, and A. Varshney, editors, *Proceedings of IEEE Visualization 2001*, pages 363–370. IEEE Computer Society Press, 2001.

[13] Peter Lindstrom and Claudio T. Silva. A Memory Insensitive Technique For Large Model Simplification. In T. Ertl, K. I. Joy, and A. Varshney, editors, *Proceedings of IEEE Visualization 2001*, pages 121–126. IEEE Computer Society Press, 2001.

[14] Zhiyan Liu, Adam Finkelstein, and Kai Li. Progressive View-Dependent Isosurface Propagation. In D. Ebert, J. M. Favre, and R. Peikert, editors, *Proceedings of the Joint Eurographics - IEEE TCVG Symposium on Visualizatation (VisSym-01)*, pages 223–232. Springer-Verlag, 2001.

[15] Y. Livnat and C. Hansen. View Dependent Isosurface Extraction. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *Proceedings of IEEE Visualization 1998*, pages 175–180. IEEE Computer Society Press, 1998.

[16] David Luebke and Carl Erikson. View-Dependent Simplification Of Arbitrary Polygonal Environments. In *SIGGRAPH 97 Conference Proceedings*, pages 199–208, Los Angeles, California. Addison Wesley.

[17] Arthur A. Mirin, Ron H. Cohen, Bruce C. Curtis, William P. Dannevik, Andris M. Dimits, Mark A. Duchaineau, D. E. Eliason, Daniel R. Schikore, S. E. Anderson, D. H. Porter, , and Paul R. Woodward. Very High Resolution Simulation Of Compressible Turbulence On The IBM-SP System. In *Proceedings of SuperComputing 1999*. (Also available as Lawrence Livermore National Laboratory technical report UCRL-MI-134237), 1999.

[18] V. Pascucci and C. L. Bajaj. Time Critical Isosurface Refinement And Smoothing. In *Proceedings of the 2000 IEEE symposium on Volume Visualization*, pages 33–42. ACM Press, 2000.

[19] Valerio Pascucci. Multi-Resolution Indexing For Out-Of-Core Adaptive Traversal Of Regular Grids. In G. Farin, H. Hagen, and B. Hamann, editors, *Proceedings of the NSF/DoE Lake Tahoe Workshop on Hierarchical Approximation and Geometric Methods for Scientific Visualization*. Springer-Verlag, Berlin, Germany, (to appear)., 2002. (Available as LLNL technical report UCRL-JC-140581).

[20] Tom Roxborough and Gregory M. Nielson. Tetrahedron Based, Least Squares, Progressive Volume Models With Application To Freehand Ultrasound Data. In T. Ertl, B. Hamann, and A. Varshney, editors, *Proceedings Visualization 2000*, pages 93–100. IEEE Computer Society Press, 2000.

[21] Zoë J. Wood, Mathieu Desbrun, Peter Schröder, and David Breen. Semi-Regular Mesh Extraction From Volumes. In T. Ertl, B. Hamann, and A. Varshney, editors, *Proceedings of IEEE Visualization 2000*, pages 275–282. IEEE Computer Society Press, 2000.

[22] Vijaya Ramachandran Xiaoyu Zhang, Chandrajit Bajaj. Paralel And Out-Of-Core View-Dependent Isocontour Visualization. In David Ebert, Pere Brunet, and Isabel Navaz, editors, *Proceedings of the Joint Eurographics - IEEE TCVG Symposium on Visualizatation (VisSym-02)*, Vienna, Austria, May 27–29 2002. Springer-Verlag.

[23] Yong Zhou, Baoquan Chen, and Arie Kaufman. Multiresolution Tetrahedral Framework For Visualizing Regular Volume Data. In *Proceedings of IEEE Visualization 1997*, pages 135–142. IEEE Computer Society Press.